Side Effect Free Dialogue Management in a Voice Enabled Procedure Browser

Manny Rayner, Beth Ann Hockey

Mail Stop T27-A2
ICSI/UCSC/NASA Ames Research Center
Moffett Field, CA 94035
mrayner@riacs.edu, bahockey@email.arc.nasa.gov

Abstract

We describe a general side-effect free dialogue management architecture suitable for command and control tasks, which extends the “update semantics” framework by including task as well as dialogue information in the information state. We show that this enables simple and elegant treatments of several dialogue management problems, including corrections, confirmations, querying of the environment, and regression testing. The architecture is discussed in the context of an implemented application, a voice enabled procedure browser for an astronautics domain.

1. Introduction

A spoken language system that carries out a command and control task typically divides into three main components: the input module, the output module, and the dialogue manager. The input module processes spoken input, and converts it into abstract utterance representations using speech recognition, semantic analysis, and other techniques. The output module processes abstract action requests, and transforms them into concrete external actions, such as speaking, updating a visual display, or moving a real or simulated robot. Mediating between the input and output modules, we have the dialogue manager (DM), which at the most basic level transforms abstract utterance representations into abstract action representations.

Most spoken language systems have some notion of context, which typically will include the preceding dialogue, the current state of the task, or both. For example, consider the reaction of a simulated robot to the command “Put it on the block”. This might include both remembering a recently mentioned object to serve as a referent for “it” (dialogue context), and looking around the current scene to find an object to serve as a referent for “the block” (task context). The DM will thus both access the current context as an input, and update it as a result of processing utterances.

In most dialogue systems, contextual information is distributed through the DM as part of the current program state. This means that processing of an input utterance involves at least some indirect side-effects, since the program state will normally be changed. If the DM makes procedure calls to the output module, there will also be direct side-effects in the form of exterior actions. As every software engineer knows, side-effects are normally considered a Bad Thing. They make it harder to design and debug systems, since they render interactions between modules opaque. The problem tends to be particularly acute when performing regression testing and evaluation; if a module’s inputs and outputs depend on side-effects, it is difficult or impossible to test that module in isolation. The upshot for spoken language systems is that it is often difficult to test the DM except in the context of the whole system.

In this paper, we will describe an architecture which directly addresses the problems outlined above, and which has been implemented in a substantial system, a voice-enabled procedure browser scheduled for initial deployment on the International Space Station (ISS) late in 2004. There are two key ideas. First, we split the DM into two pieces: a large piece, comprising nearly the whole of the code, which is completely side-effect free, and a small piece which is responsible for actually performing the actions. Second, we adopt a consistent policy about representing contexts as objects. Both discourse and task-oriented contextual information, without exception, is treated as part of the context object.

The rest of the paper is organised as follows. Section 2 presents the procedure browser application, and Section 3 gives an overview of the DM architecture. Section 4 describes how we have used the architecture to solve a number of specific dialogue management problems. Section 5 concludes.

2. The Clarissa procedure browser

Astronauts aboard the ISS spend a great deal of their time performing complex procedures. This often involves having one crew member reading the procedure aloud, while while the other crew member performs the task, an extremely expensive use of astronaut time. The Clarissa Procedure Assistant is designed to provide a cheaper alternative, whereby an automatic system navigates through the procedure under voice control, reading each step out as it is reached. There are about 75 different commands, of which the most important are those for navigation ("next", "previous", "go to step three"), accessing non-current steps ("preview step six", "read step eight point two"), recording, playing and deleting voice notes ("record voice note", "play voice note on step three point one", "delete voice note on step two"), and setting or cancelling alarms ("set alarm for five minutes from now", "cancel alarm at ten twenty one"). The current version has a vocabulary of about 260 words. Clarissa also includes a conventional GUI-style interface, closely integrated with the voice interface, so that many commands can be input in either voice or mouse/keyboard mode; since procedure tasks are typically hands- and eyes-busy, voice is however assumed to be the primary modality. The recognition and semantic analysis components of Clarissa have been described in earlier papers [1, 2]: here, we will focus on the dialogue management aspects.

The nature of the Clarissa task highlights several specific dialogue management problems. Since there is often a high level of background noise (up to 65 dB in the ISS Russian Module), recognition can frequently be error-prone, and it is important to provide efficient support for corrections. Some procedures, for example the ones for space-suit checkout, are extremely safety-critical, and another requirement is that the system should be able to navigate procedures in a mode which confirms that each procedure step has been completed before proceding to the next one. Finally, the DM frequently needs to query its environment, for example to retrieve voice notes or parameter values stored from a previous procedure run.

In the next two sections, we will show how the DM architecture helps address these issues.

3. Side effect free dialogue management

Clarissa implements a minimalist dialogue management framework, partly based on elements drawn from the TRIPS [3] and TrindiKit [4] architectures. The central concepts are those of *dialogue move*, *information state* and *dialogue action*. At the beginning of each turn, the dialogue manager is in an information state. Inputs to the dialogue manager are by definition dialogue moves, and outputs are dialogue actions. The behaviour of the dialogue manager over a turn is completely specified by an *update function* f of the form

$$f : State \times Move \rightarrow State \times Actions$$

Thus if a dialogue move is applied in a given information state, the result is a new information state and a set of zero or more dialogue actions. As in [3], the dialogue manager is bracketed between an *input manager* and an *output manager*. The input manager receives speech and other input directed to the dialogue manager, and transforms it into dialogue move format. The output manager takes dialogue actions produced by the dialogue manager, and transforms them into concrete sequences of procedure calls meaningful to other components of the system.

In the Clarissa system, most of the possible types of dialogue moves represent spoken commands. For example, `increase(volume)` represents a spoken command like "increase volume" or "speak up". Similarly, `go_to(step(2,3))` represents a spoken command like "go to step two point three". The dialogue move `undo` represents an utterance like "undo last command" or "go back" Correction utterances are represented by dialogue moves of the form `correction(X)`; so for example `correction(record(voice_note(step(4))))` represents an utterance like "no, I said record a voice note on step four". There are also dialogue moves that represent non-speech events. For example, a mouse-click on the GUI's "next" button is represented as the dialogue move `gui_request(next)`. Similarly, if an alarm goes off at time T, the message sent from the alarm agent is represented as a dialogue move of the form `alarm_triggered(T)`. The most common type of dialogue action is a term of the form `say(U)`, representing a request to speak an utterance abstractly represented by the term `U`. Other types of dialogue actions include modifying the display, changing the volume, and so on.

The information state is a vector, which in the current version of the system contains 26 elements. Some of these elements represent properties of the dialogue itself. In particular, the `last_state` element is a back-pointer to the preceding dialogue state, and the `expectations` element encodes information about how the next dialogue move is to be interpreted. For example, if a yes/no question has just been asked, the `expectations` element will contain information determining the intended interpretation of the dialogue moves `yes` and `no`.

The novel aspect of the Clarissa DM is that all *task* information is also uniformly represented as part of the information state. Thus for example the `current_location` element holds the procedure step currently being executed, the `current_alarms` element lists the set of alarms currently set, associating each alarm with a time and a message, and the `current_volume` element represents the output volume, expressed as a percentage of its maximum value. Putting the task information into the information state has the desirable consequence that actions whose effects can be defined in terms of their effect on the information state need not be specified directly. For example, the update rule for the dialogue move `go_to(Loc)` specifies among other things that the value of `current_location` element in the output dialogue state will be `Loc`. The specific rule does not also need to say that an action needs to be produced to update the GUI by scrolling to the next location; that can be left to a general rule, which relates a change in the `current_location` to a scrolling action.

More formally, what we are doing here is dividing the work performed by the update function f into two functions, g and h. g is of the same form as f, i.e.

$$g : State \times Move \rightarrow State \times Actions$$

As before, this maps the input state and the dialogue move into an output state and a set of actions; the difference is that this set now only includes the *irreversible* actions. The remaining work is done by a second function

$$h : State \times State \rightarrow Actions$$

which maps the input state S and output state S' into the set of reversible actions required to transform S into S'; the full set of output actions is the union of the reversible and the irreversible actions. The relationship between the functions f, g and h can be expressed as follows. Let S be the input state, and M the input dialogue move. Then if $g(S, M) = \langle S', A_1 \rangle$, and $h(S, S') = A_2$, we define $f(S, M)$ to be $\langle S', o(A_1 \cup A_2) \rangle$, where o is a function that maps a set of actions into an ordered sequence.

In the Clarissa system, h is implemented concretely as the set of all solutions to a Prolog predicate which contains one clause for each type of difference between states which can lead to an action. Thus we have for example a clause which says that a difference in the `current_volume` elements between the input state and the output state requires a dialogue action that sets the volume; another clause which says that an alarm time present in the `current_alarms` element of the input state but absent in the `current_alarms` element requires a dialogue action which cancels an alarm; and so on. The ordering function o is defined by a table which associates each type of dialogue action with a priority; actions are ordered by priority, with the function calls arising from the higher-priority items being executed first. Thus for example the priority table defines a `load_procedure` action as being of higher priority than a `scroll` action, capturing the requirement that the system needs to load a procedure into the GUI before it can scroll to its first step.

4. Specific issues

4.1. "Undo" and "correction" moves

As already noted, one of the key requirements for Clarissa is an ability to handle "undo" and "correction" dialogue moves. The conventional approach, as for example implemented in the CommandTalk system [5], involves keeping a "trail" of actions, together with a table of inverses which allow each action to be undone. The extended information state approach described above permits a more elegant solution to this problem, in which corrections are implemented using the g and h functions together with the `last_state` element of the information state. Thus if we write u for the "undo" move, and $l(S)$ to denote the state that S's `last_state` element points to, we can define $g(S, u)$ to be $\langle l(S), \emptyset \rangle$, and hence $f(S, u)$ will be $\langle l(S), o(h(S, l(S))) \rangle$. Similarly, if we write $c(M)$ for the move which consists of a correction followed by M, we can define $f(S, c(M))$ to be $\langle S', o(A \cup h(S, S')) \rangle$, where S' and A are defined by $g(l(S), M) = \langle S', A \rangle$.

In practical terms, there are two main payoffs to this approach. First, code for supporting undos and corrections shrinks to a few lines, and becomes trivial to maintain. Second, corrections are in general faster to execute than they would be in the conventional approach, since the h function directly computes the actions required to move from S to S', rather than first undoing the actions leading from $l(S)$ to S, and then redoing the actions from $l(S)$ to S'. When actions involve non-trivial redrawing on the visual display, this difference can be quite significant.

4.2. Confirmations

Confirmations are in a sense complementary to corrections. Rather than making it easy for the user to undo an action they have already carried out, the intent is to repeat back to them the dialogue move they appear to have made, and give them the option of not performing it at all. Confirmations can also be carried out at different levels. The simplest kind of confirmation echoes the exact words the system believed it recognised. It is usually, however, more useful to perform confirmations at a level which involves further processing of the input. This allows the user to base their decision about whether to proceed not merely on the words the system believed it heard, but also on the actions it proposes to take in response.

The information state framework also makes possible a simple approach to confirmations. Here, the key idea is to compare the current state with the state that would arise after responding to the proposed move, and repeat back a description of the difference between the two states to the user. To write this symbolically, we start by introducing a new function $d(S, S')$, which denotes a speech action describing the difference between S and S', and write the dialogue moves representing "yes" and "no" as y and n respectively. We can then define $f_{conf}(S, M)$, a version of $f(S, M)$ which performs confirmations, as follows. Suppose we have $f(S, M) = \langle S', A \rangle$. We define $f_{conf}(S, M)$ to be $\langle S_{conf}, d(S, S') \rangle$, where S_{conf} is constructed so that $f(S_{conf}, y) = \langle S', A \rangle$ and $f(S_{conf}, n) = \langle S, \emptyset \rangle$. In other words, S_{conf} is by construction a state where a "yes" will have the same effect as M would have had on S if the DM had proceeded directly without asking for a confirmation, and where a "no" will leave the DM in the same state as it was before receiving M.

There are two points worth noting here. First, it is easy to define the function f_{conf} precisely because f is side-effect free; this lets us derive and reason about the *hypothetical* state S' without performing any external actions. Second, the function $d(S, S')$ will in general be tailored to the requirements of the task, and will describe *relevant* differences between S and S'. In Clarissa, where the critical issue is which procedure steps have been completed, $d(S, S')$ describes the difference between S and S' in these terms, for example saying that one more step has been completed, or three steps skipped.

4.3. Querying the environment

An obvious problem for any side-effect free dialogue management approach arises from the issue of querying the environment. If the DM needs to acquire external information to complete a response, it may seem that the relationship between inputs and output can no longer be specified as a self-contained function.

The framework can, however, be kept declarative by splitting up the DM's response into two turns. Suppose that the DM needs to read a data file in order to respond to the user's query. The first turn responds to the user query by producing an action request to read the file and report back to the DM, and an output information state in which the DM is waiting for a dialogue move reporting the contents of the file; the second turn responds to the file-contents reporting action by using the new information to reply to the user. The actual side-effect of reading the file occurs outside the DM, in the space between the end of the first turn and the start of the second. Variants of this scheme can be applied to other cases in which the DM needs to acquire external information.

4.4. Regression testing and evaluation

Regression testing and evaluation on context-dependent dialogue systems is a notoriously messy task. The problem is that it is difficult to assemble a reliable test library, since the response to each individual utterance is in general dependent on the context produced by the preceding utterances. If an utterance early in the sequence produces an unexpected result, it is usually impossible to know whether results for subsequent utterances are meaningful.

In our framework, regression testing of contextually dependent dialogue turns is unproblematic, since the input and output contexts are well-defined objects. We have been able to construct substantial libraries of test examples, where each example consists of a 4-tuple ⟨InState, DialogueMove, OutState, Actions⟩. These libraries remain stable over most system changes, except for occasional non-downward-compatible redesigns of the dialogue context format, and have proved very useful.

5. Summary and conclusions

We have described a general side-effect free dialogue management architecture suitable for command and control tasks, which extends the "update semantics" framework by including task as well as dialogue information in the information state. We have shown that this enables simple and elegant treatments of several dialogue management problems, including corrections, confirmations, querying of the environment, and regression testing. The methods have been implemented in a non-trivial application, and have performed well there.

The main obstacle to general applicability is that the approach relies on being able to represent the task state completely. For many applications, like the one described here, this is however entirely possible, and in these cases the approach appears extremely effective.

6. Acknowledgements

The greater part of the work described here was carried out at the Research Institute for Advanced Computer Science, NASA Ames Research Center, under Cooperative Agreement NCC 2–1006.

7. References

[1] G. Aist, J. Dowding, B. Hockey, and J. Hieronymus, "An intelligent procedure assistant for astronaut training and support," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (demo track)*, Philadelphia, PA, 2002.

[2] M. Rayner, B. A. Hockey, J. Hieronymus, J. Dowding, and G. Aist, "An intelligent procedure assistant built using REGULUS 2 and ALTERF," in *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (demo track)*, Sapporo, Japan, 2003.

[3] J. Allen, D. Byron, M. Dzikovska, G. Ferguson, L. Galescu, and A. Stent, "An architecture for a generic dialogue shell," *Natural Language Engineering, Special Issue on Best Practice in Spoken Language Dialogue Systems Engineering*, pp. 1–16, 2000.

[4] S. Larsson and D. Traum, "Information state and dialogue management in the TRINDI dialogue move engine toolkit," *Natural Language Engineering, Special Issue on Best Practice in Spoken Language Dialogue Systems Engineering*, pp. 323–340, 2000.

[5] A. Stent, J. Dowding, J. Gawron, E. Bratt, and R. Moore, "The CommandTalk spoken dialogue system," in *Proceedings of the Thirty-Seventh Annual Meeting of the Association for Computational Linguistics*, 1999, pp. 183–190.